МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 0304 _____ Алексеев Р.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2022

Цель работы.

Изучить алгоритм Ахо-Корасик для нахождения вхождений подстрок в строку.

Реализовать алгоритм Ахо-Корасик.

Задание.

Вариант 2

Подсчитать количество вершин в автомате; вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.

Задание 1

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T, $1 \le |T| \le 100000$).

Вторая - число n ($1 \le n \le 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{ p_1, \ldots, p_n \}$ $1 \le |p_i| \le 75$

Все строки содержат символы из алфавита {*A, C, G, T, N*}

Выход:

Все вхождения образцов из P в T.

Каждое вхождение образца в текст представить в виде двух чисел - i p

Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p (нумерация образцов начинается с 1).

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

Sample Input:

NTAG

3

TAGT

TAG

T

Sample Output:

2 2

2 3

Задание 2

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с джокером.

В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу *P* необходимо найти все вхождения *P* в текст *T.*

Например, образец *ab??c?* с джокером *?* встречается дважды в тексте *xabvccbababcax*.

Символ джокер не входит в алфавит, символы которого используются в *T*. Каждый джокер соответствует одному символу, а не подстроке неопределённой длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида *???* недопустимы.

Все строки содержат символы из алфавита {*A, C, G, T, N*}

Вход:

Текст (*T*, $1 \leq |T| \leq 100000$)

Шаблон (*P*, $1 \leq |P| \leq 40$)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Sample Input:

ACTANCA

A$$A$

$

Sample Output:

1

Выполнение работы.

Описание алгоритма

Алгоритм Ахо-Корасик находит все вхождения набора шаблонов в строке, используя автомат.

Автомат получает по очереди все символы строки и переходит по соответствующим ребрам, если он пришел в конечное состояние, значит шаблон является подстрокой строки.

Все шаблоны из набора шаблонов объединяютя в дерево поиска — бор. Построение бора осуществляется в несколько этапов:

1. Выбирается очередной шаблон из набора.
2. Символы шаблона, начиная с первого, сравниваются с символами на исходящих из текущей вершины ребрах.
3. Если найдено совпадаение, то переходим к новой вершине по соответсующему ребру, после чего повторяем шаг 2 пока не закончится шаблон или не будет найдено совпадение символов.
4. Если соответствующего символа обнаружено не было, то строится новая вершина, после чего повторяется шаг 2.
5. Если символы в шаблоне кончились, текущая вершина становится конечной для шаблона.

Для нахождения вхождений шаблонов в строку используются суффиксные ссылки. Суффиксная ссылка — это ссылка на вершину, соответсвующую самому длинному суффиксу или, если такого нет, корню. Для корневого узла суффиксная ссылка — петля на самого себя. Для остальных узлов переходим по суффкисной ссылке родителя и проверяем, есть ли из новой вершины ребра с требуемым символом, если да, то вершина, куда ведет ребро, становится суффиксной ссылкой изначальной вершины, в противном случае повторяем переходы, пока не найдем подходящую вершину или не попадем в корень. Для ускорения работы алгоритма используются «хороши» суффиксные ссылки, т. е. суффиксные ссылки, ведущие к наибольшим суффиксам, которые являются концами шаблонов.

Алгоритм Ахо-Корасик проходит по всем символам исходной строки, для каждого символа переходит по «хорошим» суффиксным ссылкам в боре, учитывая позицию символа в исходной строке.

Для решения первой задачи использовался алгоритм Ахо-Корасик, описанный выше. Для решения второй задачи шаблона с джокерами, где джокер — это символ, которого нет в алфавите, на месте которого может быть любой символ алфавита, разделялся на набор шаблонов при помощи

удаления всех джокеров. Для полученного набора шаблонов применялся алгоритм Ахо-Корасик, опсианный выше, в качестве выходных данных получался список, *i*-му символу которого соответсвовало число шаблонов, начинающихся в исходной строке с этого символа, с учетом позиции шаблона в шаблоне с джокерами. Если *i*-й элемент равен количеству шаблонов, значит с него начинается вхождение шаблона с джокерами.

Сложность алгоритма

Сложность построения бора по количеству операций — O(*m*), где *m* — суммарная длина всех шаблонов, т. к. необходимо пройти по каждому символу каждого шаблона, внося их в бор. Сложность по памяти — O(*m*), т. к. в худшем случае для каждого символа будет выделяться свое ребро, следовательно и вершина.

Для построения суффиксных ссылок производится обход бора в ширину, для каждой конкретной вершины необходимо перейти по суффиксной ссылке родителя, поэтому сложность по числу операций — O(1), по памяти — O(1). Т.к. необходимо построить суффиксные ссылки для каждой вершины бора, то суммарная сложность по числу операций — O(*m*), по памяти — O(1), т. к. дополнительной памяти не требуют, т. к. храняться в вершнинах бора.

Для построения «хороших» суффиксных ссылок используются обычные суффиксные ссылки. Для получения конкретной «хорошей» суффиксной ссылки необходимо переходить по суффиксным ссылкам от вершины пока не будет найдена либо вершина, являющаяся конечной в шаблоне, либо не будет найден корен. Т.к. необходимо построить «хорошие» суффиксные ссылки для каждой вершины бора, то сложность по числу операций — O(*m*), т. к. дополнительная память не требуется, то сложность по памяти — O(1).

Для получения функции перехода между состояниями для конкретной вершины требует в худшем случае перехода по всем суффиксным ссылками от вершины до корня, поэтому в худшем случае сложность по числу операций — O(m), т. к. функция перехода храниться в вершине бора, то дополнительная память не требуется, следовательно сложность по памяти — O(1). Т.к. каждой вершине необходимо получить функцию перехода для каждого символа алфавита, то сложность по числу операций — O(mk), по памяти — O(mk), где k — число символов в алфавите.

Для нахождения всех вхождений шаблонов в строку алгоритм Ахо-Корасик проходит по каждому символу исходной строки, переходит к соответсвующей вершине и переходит по «хорошим» суффиксным ссылкам пока не достигнет корня. Т.к. прохождение осуществляется только по «хорошим» суффиксным ссылкам, т. е. суффиксными ссылками, ведущими в вершины, которые являются концами шаблонов, то сложность по числу операций — O($n + t$), где n — количество символов в исходной строке, t — число вхождений шаблонов в исходную строку. Т.к. дополнительная память не требуется, то сложность по памяти — O(1).

Суммарная сложность по числу операций будет O(m) + O(m) + O(m) + O(mk) + O($n + t$) = O($n + mk + t$), по памяти — O(m) + O(1) + O(1) + O(mk) + O(1) = O(mk).

При поиске вхождений шаблона с джокерами необходимо сначала разделить шаблон на набор подстрок без джокеров, для этого осуществляется проход по шаблону и сравнивание каждого символа, следовательно сложность разделения по чилу операций — O(m), по памяти — O(m), т. к. в худшем случае в шаблоне нет ни одного джокера. Далее используется алгоритм Ахо-Корасик, сложность которого по числу операций — O($n + mk + t$), по памяти — O(m). Результат работы алгоритма сохраняется в список длины равной длине исходной строки, следовательно сложность по памяти

становится — O(n + m). Для нахождения всех вхождений производится проход по полученному списку и сравнение значений с количеством подстрок, следовательно сложность по числу операций — O(n), по памяти — O(1). Суммарная сложность по числу операций — O(m) + O(n + mk + t) + O(n) = O(n + mk + t), по памяти — O(mk) + O(n + m) + O(1) = O(n + mk).

Функции и структуры данных

Для создания вершин бора был создан класс *BohrTop*. Класс имеет восемь полей: *self.pattern* — список шаблонов, заканчивающихся этой вершиной, *self.nextTop* — список вершин-детей, *self.flag* — флаг того, является ли вершина концом шаблона, *self.suffLink* — суффиксная ссылка, *self.suffFLink* - «хорошая» суффиксная ссылка, *self.autoMove* — список функций перехода, *self.par* — ссылка на предка, *self.symb* — номер символа на ребере, ведущем в вершину. Для вывода вершины бора перегружен метод *__str__(self)*, метод возвращает номер символа в виде строки.

Для хранения данных и их обработки создан класс *AlgAC*.

Класс имеет двенадцать полей: *self.strT* — исходная строка, *self.setP* — список шаблонов, *self.result* — список вхождений шаблонов в исходную строку, *self.patterns* — список шаблонов, добавленных в бор, *self.bohr* — бор, *self.alphabet* — список символов алфавита, *self.isJocker* — флаг того, какое из заданий выполняется, *self.interConc* — флаг для промежуточных выводов, *self.numP* — список индексов начал вхождений подстрок без джокеров в шаблон с джокерами, *self.c* — список количеств начал вхождений подстрок на *i* -ом символе исходной строки, *self.resultJocker* — список вхождений шаблона с джокерами в исходную строку, *self.lenPat* — длина шаблона с джокерами.

Для вывода промежуточных выводов и результатов, для обработки данных реализован ряд методов.

Метод *printBohr(self)* используется для начала вывода бора. Метод выводит первую вершину бора и запускает рекурсивный вывод остальных вершин вызывая метод *printTopBohr(self, top, pref, isLast).*

Метод *printTopBohr(self, top, pref, isLast)* используется для рекурсивного вывода вершин бора. Метод принимает: *top* — номер вершины, *pref* — префикс, который необходимо вывести перед вершиной, *isLast* — флаг того, является ли вершина последней. При вызове метод выводит префикс и ребро с символом, после чего рекурсивно вызывает себя для каждой вершины-ребенка. В зависимости от того, является вершина последней или нет, префикс увеличивается на опредленный набор символов. Если у вершины нет детей, то глубина рекурсии уменьшается.

Метод *printResult(self)* используется для вывода результатов работы алгоритма Ахо-Корасик. Метод сортирует результаты по возрастанию, после чего выводит их на экран.

Метод *printInd(self)* используется для вывода индивидуализации. Метод выводит на экран количетсво вершин в боре и проходит по результатам работы алгоритма Ахо-Корасик и проверяет шаблоны, входящие в исходную строку, на то, пересекаются ли они между собой в исходной строке, пересекающиеся шаблоны выводятся на экран.

Метод *createBohr(self)* используется для создания бора. Метод проходит по списку шаблонов и для каждого вызывает метод добавления строки в бор — *addStrToBohr(self, newStr).*

Метод *setNumLen(self, numP, lenPat)* используется для сохранения дополнительных данных. Метод принимает: *numP* — список индексов начал подстрок в шаблоне с джокерами, *lenPat* — длина шаблона. Метод сохраняет список индексов начал подстрок в шаблоне с джокерами и длину шаблона, а также ставит флаг, что выполняется поиск вхождений шаблона с джокерами.

Метод *addStrToBohr(self, newStr)* используется для добавления строки в бор. Метод принимает: *newStr* — строка, которую необходимо добавить в бор. Метод проходит по символам строки, проверяя, есть ли исходящее ребро с необходимым символом, если есть, то переходит по нему и обрабатывает следующий символ, в противном случае добавляет новое ребро и новую вершину. Для последний вершины ставит флаг конца шаблона в значение *True* и добавляет шаблон в список шаблонов вершины.

Метод *getAutoMove(self, v, ch)* используется для получения функций перехода. Метод получает: *v* — номер вершины, *ch* — номер символа. Если для соответствующего символа у вершины нет функции перехода, то проверяется наличие исходящего ребра с этим символом. Если ребро есть, то ссылка на вершину, в которую оно ведет, устанавливается как функция перехода, в противном случае, если вершина корень, то функция указывает на корень, иначе ищется функция перехода для вершины по суффиксной ссылке. Метод возвращает функцию перехода, являющуюся номером вершины, в которую необходимо перейти.

Метод *getSuffLink(self, v)* используется для получения суффиксной ссылки. Метод принимает: *v* — номер вершины. В случае, если суффиксная ссылка ещё не була установлена, если вершина является корнем или её родитель корень, то суффиксная ссылка устанавливается на корень, в противном случае устанавливается на функцию перехода для вершины по суффиксной ссылке родителя. Метод возвращает суффиксную ссылку.

Метод *getSuffFLink(self, v)* используется для получения «хорошей» суффиксной ссылки. Метод принмиает: *v* — номер вершины. Если «хорошая» суффиксная ссылка не была установлена, то находится обычная суффиксная ссылка. Если она ведет к корню, то «хорошая» суффиксная ссылка устанавливается на корень, иначе проверяется на то, является ли вершина по суффиксной ссылке концом шаблона, если да, то «хорошая» суффиксная

ссулка устанавливается на неё, иначе устанавливается на «хорошую» суффиксную ссылку для этой вершины. Метод возвращает «хорошую» суффиксную ссылку.

Метод *search(self, u, i)* используется для нахождения вхождений шаблонов, оканчивающихся на *u*-й вершине. Метод принимает: *u* — номер вершины, *i* — индекс символа в исходной строке. Метод проходит по хорошим суффиксным ссылкам, пока не дойдет до корня. Найденные вхождения шаблонов сохраняются в поля *self.result* и *self.c*.

Метод *searchAll(self)* используется для нахождения всех вхождений шаблонов в исходную строку. Метод проходит по всем символам исходной строки и для каждого из них вызывает метод *search(self, u, i)*.

Метод *searchJocker(self)* используется для нахождения вхождений шаблона с джокерами. Метод вызвает метод *searchAll(self)* для нахождения всех вхождений, после чего проходит по списку в поле *self.c*. Если *i*-й элемент равен количеству подстрок в шаблоне с джокерами, то этот индекс сохраняется в результат в поле *self.resultJocker*.

Для разделения шаблона с джокерами на набр подстрок без джокеров создана функция *cutPattern(pattern, jocker)*. Функция принимает: *pattern* — шаблон с джокерами, *jocker* — символ джокера. Функция проходит по шаблону, сохраняя в список подстроки без джокера. Функция возвращает список подстрок без джокеров и список индексов, с которых эти подстроки начинаются в шаблоне.

Для решения первой задачи создана функция *first(interConc)*. Функция принимает: *interConc* — флаг промежуточных выводов. Функция принимает данные с клавиатуры и вызывает необходимые методы класса *AlgAC*.

Для решения второй задачи создана функция *second(interConc)*. Функция принимает: *interConc* — флаг промежуточных выводов. Функция

принимает данные с клавиатуры и вызывает необходимые методы класса *AlgAC*.

Индивидуализация

Для входных данных:

ACCACCAAC

4

ACC

ACA

AAC

CAA

Был построен бор, результат на рис. 1.

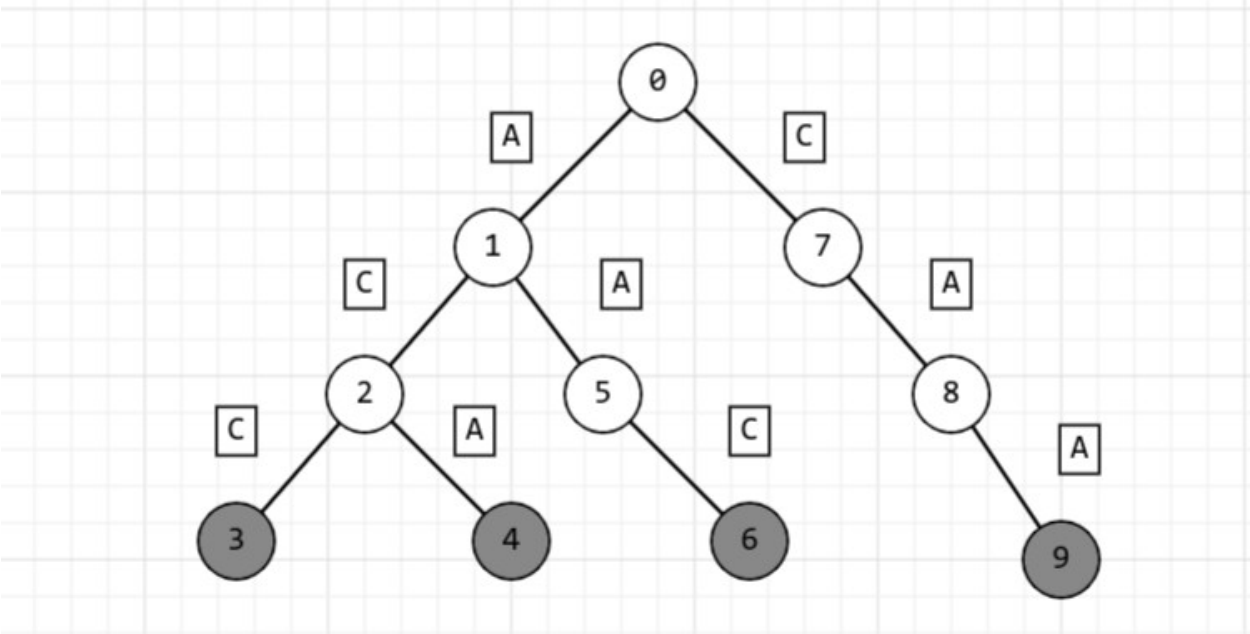

Рисунок 1 — Бор (Серые вершины — концы шаблонов)

По рисунку видно, что в боре 10 вершин.

Также были найдены пересекающиеся в исходной строке шаблоны: ACC с CAA и CAA с AAC. После чего была запущена программа, результат представлен на рис. 2.

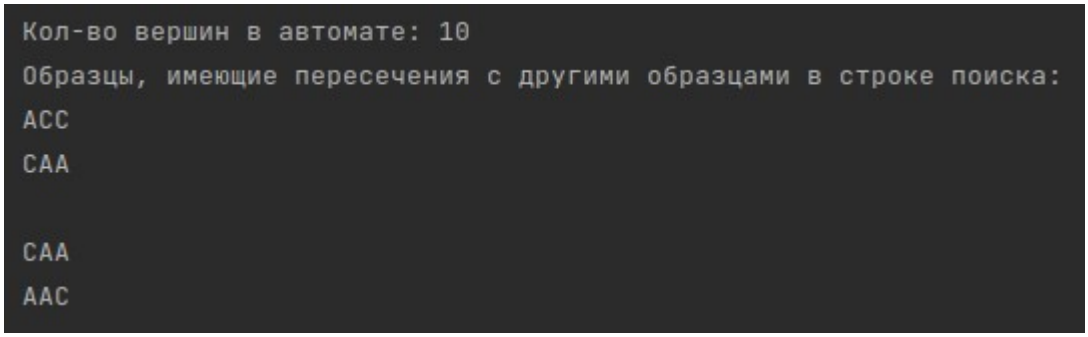

Рисунок 2 — Результат программы.

По рисунку видно, что результаты совпали с теорретическими, следовательно программа работает правильно.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Выводы.

Был изучен алгоритм Ахо-Корасик для нахождения всех вхождений подстрок в строку.

Была написана программа, находящая вхождения набора шаблонов в строку при помощи алогоритма Ахо-Корасик, также программа была дополнена таким образом, чтобы она находила вхождения шаблона с джокерами при помощи алгоритма Ахо-Корасик.

Было установлено, что сложность алгоритма Ахо-Корасик по числу операций — O($n + mk + t$), по памяти — O($m + k$), где n — длина исходной строки, m — суммарная длина шаблонов, t — количество вхождений шаблонов в строку, k — число символов в алфавите. Для нахождения шаблона с джокерами сложность по числу операций - O($n + mk + t$), по памяти - O($n + m + k$).

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class BohrTop:      # Вершина бора
    def __init__(self, p, symb):
         self.pattern = []            # Номера шаблонов, которые кончаются этой вершиной
          self.nextTop = [-1] * 5      # Вершины в которые можно перейти
        self.flag = False           # Конец шаблона
        self.suffLink = -1          # Суфф. ссылка
        self.suffFLink = -1         # "Хорошая" суфф. ссылка
         self.autoMove = [-1] * 5     # Возможные переходы между состояниями
        self.par = p                # Номер предка
        self.symb = symb            # Номер символа в алфавите

    def __str__(self):
        return str(self.symb)


class AlgAC:
    def __init__(self, strT, setP, interConc):
        self.strT = strT  # Исходная строка
        self.setP = setP  # Шаблоны
        self.result = []    # Результаты
        self.patterns = []  # Шаблоны
        self.bohr = []      # Бор
        self.alphabet = ['A', 'C', 'G', 'T', 'N']   # Алфавит
        self.bohr.append(BohrTop(0, '$'))   # Инициализация бора
        self.isJocker = False   # Флаг задания
        self.interConc = interConc  # Флаг промежуточных выводов

        # Для джокера
        self.numP = []    # Индексы начал шаблонов
         self.c = [0] * len(strT)    # Список количеств вхождений по i индексу
        self.resultJocker = []  # Результаты 2 задания
        self.lenPat = 0     # Длина шаблона

    def printBohr(self):    # Вывод бора
        print("└──$──○")    # Корневая вершина
          for j in range(len(self.alphabet)): # Проход по детям корневой вершины
              isLast = True
                  for e in range(j+1, len(self.alphabet)):     # Проверка на то, является ли ребенок последним
                    if self.bohr[0].nextTop[e] != -1:
                        isLast = False
                if self.bohr[0].nextTop[j] != -1:   # Если ребенок есть
```

```
                            self.printTopBohr(self.bohr[0].nextTop[j], "
", isLast)
            print()

          def printTopBohr(self, top, pref, isLast):  # Вывод вершин
бора
            print(pref, end="")
              print(f"└──{self.alphabet[int(self.bohr[top].symb)]}──",
end='')    # Вывод ребра с символом
            if self.bohr[top].flag:
                print("  ")
            else:
                print("○")

                               if   self.bohr[top].nextTop.count(-1)   ==
len(self.bohr[top].nextTop):     # Если вершина - лист
                return

            if isLast:  # Увеличение префикса
                pref += "      "
            else:
                pref += "│     "

               for j in range(len(self.alphabet)):  # Перебор детей
вершины
                isLast = True
                     for e in range(j+1, len(self.alphabet)):     #
Проверка на то, является ли ребенок последним
                    if self.bohr[top].nextTop[e] != -1:
                        isLast = False
                  if self.bohr[top].nextTop[j] != -1: # Если ребенок
есть
                           self.printTopBohr(self.bohr[top].nextTop[j],
pref, isLast)


        def printResult(self):  # Вывод результата АК
            if self.interConc:
                print("Результат:")
            self.result.sort()
            for el in self.result:
                print(el[0], el[1])

        def printResJocker(self):   # Вывод результата АК с джокерами
            if self.interConc:
                print("Результат:")
            self.resultJocker.sort()
            for el in self.resultJocker:
                print(el)

        def printInd(self):     # Вывод индивидуализации
            print(f"\nКол-во вершин в автомате: {len(self.bohr)}")
            intersec = []
            '''for i in range(len(self.result)):
                for e in range(i, len(self.result)):
```

```
                    if self.result[e] != self.result[i]:
                                                    if  self.result[i][0]  +
len(self.setP[self.result[i][1]-1]) > self.result[e][0]:
                                intersec.append([self.setP[self.result[i]
[1]-1], self.setP[self.result[e][1]-1],
                                                        self.result[i][0]-1,
self.result[e][0]-1])
            if len(intersec):
                        print("Образцы, имеющие пересечения с другими
образцами в строке поиска:")
                for el in intersec:
                     print(f"\tОбразцы {el[0]} и {el[1]} пересекаются
символами под номерами:")
                    diff = el[2] + len(el[0]) - el[3]
                    if diff >= len(el[1]):
                        lenIntersec = len(el[1])
                    else:
                        lenIntersec = diff
                    for i in range(lenIntersec):
                              print("\t", el[3] + i + 1, f"- символ
{self.strT[el[3] + i]}")
                    print()
            else:
                  print("Образцы не пересекаются между собой в строке
поиска")'''

            for i in range(len(self.result)):
                for e in range(i, len(self.result)):
                      if self.result[e] != self.result[i]:    # Если
результаты различны
                                                    if  self.result[i][0]  +
len(self.setP[self.result[i][1]-1]) > self.result[e][0]:
                                intersec.append(self.setP[self.result
[i][1] - 1])
                                intersec.append(self.setP[self.result
[e][1] - 1])
                                intersec.append("")
            if len(intersec):
                        print("Образцы, имеющие пересечения с другими
образцами в строке поиска:")
                for el in intersec:
                    print(el)
            else:
                  print("Образцы не пересекаются между собой в строке
поиска")

        def createBohr(self):   # Создание бора
            if self.interConc:
                print("Создание бора:")
            for p in self.setP:     # Перебор всех шаблонов в списке
                if self.interConc:
                    print(f"Добавим в бор шаблон {p}:")
                    print("Бор до добавления:")
                    self.printBohr()
                self.addStrToBohr(p)    # Добавление шаблона
```

```
                if self.interConc:
                        print(f"Бор в результате добавления шаблона
{p}:")
                    self.printBohr()

        def setNumLen(self, numP, lenPat):  # Сохранение доп. данных
            self.numP = numP
            self.lenPat = lenPat
            self.isJocker = True

        def addStrToBohr(self, newStr):  # Добавление новой строки в
бор
            num = 0  # Номер вершины бора
            for i in range(len(newStr)):
                numChar = self.alphabet.index(newStr[i])  # Номер
символа в алфавите
                if self.bohr[num].nextTop[numChar] == -1:  # Ребра
нет
                    self.bohr.append(BohrTop(num, numChar))
                    self.bohr[num].nextTop[numChar] = len(self.bohr)
- 1
                    if self.interConc:
                            print(f"Символа {newStr[i]} под индексом
{i+1} нет в боре, добавляем новую вершину. Бор после "
                                  f"добавления:")
                        self.printBohr()
                elif self.interConc:
                    print(f"Символ {newStr[i]} под индексом {i+1} уже
есть в боре, переходим к след. символу шаблона")
                num = self.bohr[num].nextTop[numChar]
            self.bohr[num].flag = True  # Вершина является последней
для шаблона
            self.patterns.append(newStr)
            self.bohr[num].pattern.append(len(self.patterns) - 1)  #
Сохранения номера шаблона, заканчиающегося вершиной

        def getAutoMove(self, v, ch):   # Получение функции перехода
между состояниями
            if self.bohr[v].autoMove[ch] == -1:      # Если функции
перехода ещё нет
                if self.bohr[v].nextTop[ch] != -1:  # Если есть дети
                                        self.bohr[v].autoMove[ch]  =
self.bohr[v].nextTop[ch]
                else:
                    if v == 0:   # Если вершина - корень
                        self.bohr[v].autoMove[ch] = 0
                    else:
                                         self.bohr[v].autoMove[ch]  =
self.getAutoMove(self.getSuffLink(v), ch)
            return self.bohr[v].autoMove[ch]

        def getSuffLink(self, v):  # Получение суффиксной ссылки для
v вершины
            if self.bohr[v].suffLink == -1:  # Ещё не нашли суфф.
ссылку
```

```
                if self.interConc:
                    print(f"Для вершины {v} ещё нет суфф. ссылки")
                if v == 0 or self.bohr[v].par == 0:  # V или предок V
- корень
                    if self.interConc:
                           print(f"Вершина {v} является корнем или её
предок корень, суфф. ссылка устанавливается на корень")
                    self.bohr[v].suffLink = 0
                else:
                                                self.bohr[v].suffLink   =
self.getAutoMove(self.getSuffLink(self.bohr[v].par),
self.bohr[v].symb)
                    if self.interConc:
                                  print(f"Суфф. ссылка для вершины {v}
устанавливается на вершину {self.bohr[v].suffLink}")
            elif self.interConc:
                     print(f"Суфф. ссылка для вершины {v} уже была
установлена - {self.bohr[v].suffLink}")
            if self.interConc:
                print()
            return self.bohr[v].suffLink

        def getSuffFLink(self, v):   # Получение "хорошей" суфф.
ссылки
            if self.bohr[v].suffFLink == -1:  # Ещё не нашли суфф.
ссылку
                if self.interConc:
                       print(f"Для вершины {v} ещё нет хорошей суфф.
ссылки")
                u = self.getSuffLink(v)
                if u == 0:  # V - корень, или U указывает на корень
                    if self.interConc:
                           print(f"Вершина {v} является корнем или её
предок корень, хорошая суфф. ссылка устанавливается "
                                 f"на корень")
                    self.bohr[v].suffFLink = 0
                else:
                    if self.bohr[u].flag:
                        self.bohr[v].suffFLink = u
                    else:
                        self.bohr[v].suffFLink = self.getSuffFLink(u)
                        if self.interConc:
                             print(f"Хорошая суфф. ссылка для вершины
{v} устанавливается на вершину "
                                   f"{self.bohr[v].suffFLink}")
            elif self.interConc:
                print(f"Хорошая суфф. ссылка для вершины {v} уже была
установлена - {self.bohr[v].suffFLink}")
            if self.interConc:
                print()
            return self.bohr[v].suffFLink

        def search(self, u, i):      # Поиск вхождений шаблона по
индексу
            if self.interConc:
```

```
                print(f"Поиск для вершины {u} и символа под индексом
{i}")
            while u != 0:
                    if self.bohr[u].flag:     # Если в вершине
заканчивается шаблон
                    if self.interConc:
                        print(f"Вершина {u} является концом для
одного из шаблонов в боре")
                    numberPat = self.bohr[u].pattern
                    for pat in numberPat:
                        lenPat = len(self.patterns[pat])
                        if self.interConc:
                            print(f"Сохранение в результат шаблона
{self.setP[pat]} и индекса начала его"
                                  f"вхождения {i - lenPat + 1}")
                        self.result.append([i - lenPat + 1, pat + 1])
                        # Для джокера
                        if self.isJocker and i - self.numP[pat] >= 0:
                            if self.interConc:
                                print(f"Увеличение кол-ва шаблонов
без джокеров, начинающихся с индекса "
                                      f"{i - lenPat -
self.numP[pat]}")
                            self.c[i - self.numP[pat] - lenPat] += 1
                u = self.getSuffFLink(u)
                if self.interConc:
                    print(f"Переходим по \"хорошей\" суффиксной
ссылке к вершине {u}")
                if self.interConc:
                    print(f"Переход по суффиксной ссылке к новой
вершине - {u}")
            if self.interConc:
                print("По суффиксной ссылке пришли к корню, дальнешие
переходы не имеют смысла\n")

        def searchAll(self):    # Поиск вхождений всех шаблонов
            if self.interConc:
                print("Поиск вхождений шаблонов в строку поиска:")
            u = 0
              for i in range(len(self.strT)):      # Проход по всем
символам строки
                if self.interConc:
                    print(f"Для вершны {u} и символа {self.strT[i]}
переходим к вершине", end=" ")
                                        u   =   self.getAutoMove(u,
self.alphabet.index(self.strT[i]))
                if self.interConc:
                    print(f"{u}")
                self.search(u, i+1)
            if self.interConc:
                print()

        def searchJocker(self):     # Поиск с джокерами
            if self.interConc:
                print("Поиск вхождений шаблона с джокерами")
```

```
            self.searchAll()
            for i in range(len(self.c)):
                if self.c[i] == len(self.setP) and i + self.lenPat <=
len(self.strT):
                    if self.interConc:
                          print(f"Сохранение индекса начала вхождения
шаблона - {i + 1}")
                    self.resultJocker.append(i + 1)
            if self.interConc:
                print()


    def cutPattern(pattern, jocker):  # Нарезка шаблона на подстроки
без джокеров
        setP = []
        numP = []
        buf = ''
        flag = False
        e = 0
        for i in range(len(pattern)):   # Проход по символам шаблона
            if pattern[i] == jocker:    # Если символ - джокер
                if flag:
                    setP.append(buf)
                    buf = ''
                    numP.append(e)
                flag = False
            else:
                if not flag:
                    e = i
                buf += pattern[i]
                flag = True
        if buf != '':
            setP.append(buf)
            numP.append(e)
        return setP, numP


    def first(interConc):    # Первая часть
        strT = input()
        n = int(input())
        setP = []
        for _ in range(n):  # Считывание шаблонов
            setP.append(input())
        alg = AlgAC(strT, setP, interConc)
        alg.createBohr()
        alg.searchAll()
        alg.printResult()
        alg.printInd()


    def second(interConc):   # Вторая часть
        strT = input()
        pattern = input()
        jocker = input()
```

```
        setP, numP = cutPattern(pattern, jocker)
        if interConc:
               print("Шаблон был разделен на подстроки без джокеров,
результат:")
            for i in range(len(setP)):
                print(numP[i], setP[i])
            print()
        lenPat = len(pattern)

        alg = AlgAC(strT, setP, interConc)
        alg.setNumLen(numP, lenPat)
        alg.createBohr()
        alg.searchJocker()
        alg.printResJocker()
        #alg.printInd()


    if __name__ == '__main__':
        first(True)
        #second(True)
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Таблица Б.1 - Тестирование для 1 задания

№ п/п	Входные данные	Выходные данные	Комментарии
1.	ACCACCAAC 4 ACC ACA AAC CAA	1 1 4 1 6 4 7 3	
2.	AAAAA 2 A AA	1 1 1 2 2 1 2 2 3 1 3 2 4 1 4 2 5 1	
3.	A 1 A	1 1	
4.	NTAG 3 TAGT TAG T	2 2 2 3	
5.	NTAAATN 4 AA AA NTA NTAAATN	1 3 1 4 3 1 3 2 4 1 4 2	

6.	AAATAAN 3 AANT GA AAATAAC		

Таблица Б.2 — Тестирование для 2 задания

№ п/п	Входные данные	Выходные данные	Комментарии
1.	ACTANCA A$$A$ $	1	
2.	ACCACCAACACAA $$A$ $	1 4	
3.	ACTAAGCCCATCC NC A$$C$ $	4 5 10	
4.	AAAAAAAA AA$$AA $	1 2 3	
5.	AGAGAGAGAG $$$$$$$$$G $	1	
6.	AAAAAAAAAA $$$$A$$$$ $	1 2	
7.	AGAGAGAGAG $$$T $		

Таблица Б.3 — Тестирование индивидуализации

№ п/п	Входные данные	Выходные данные	Комментарии
1.	ACCACCAAC 4 ACC ACA AAC CAA	10 ACC CAA CAA AAC	
2.	AAAAA 2 A AA	3 A AA AA A AA AA A AA AA A AA AA A AA AA	

		A AA AA A AA AA A	
3.	A 1 A	2	
4.	NTAG 3 TAGT TAG T	5 TAG T	
5.	NTAAATN 4 AA AA NTA NTAAATN	10 NTA AA NTA AA NTAAATN AA NTAAATN AA NTAAATN	

		AA NTAAATN AA AA AA AA AA AA AA AA AA AA AA AA AA	
6.	AAATAAN 3 AANT GA AAATAAC	12	